WEST

End of Result Set

Generate Collection Print

L3: Entry 1 of 1 File: USPT Feb 26, 2002

US-PAT-NO: 6351845
DOCUMENT-IDENTIFIER: US 6351845 B1

TITLE: Methods, apparatus, and articles of manufacture for analyzing memory use

DATE-ISSUED: February 26, 2002

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Hinker; Paul	Longmont	CO		
Dennie; Shaun	Westminster	CO		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
Sun Microsystems, Inc.	Palo Alto	CA			02

APPL-NO: 09/ 244894 [PALM]
DATE FILED: February 4, 1999

trace file

INT-CL: [07] G06 F 9/45

US-CL-ISSUED: 717/4; 711/100
US-CL-CURRENT: 717/128; 711/100

FIELD-OF-SEARCH: 717/9, 717/4, 717/8, 711/100

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

Search Selected Search ALL

	PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
☐	5613063	March 1997	Eustace et al.	714/38
☐	5689712	November 1997	Heisch	395/704
☐	5787480	July 1998	Scales et al.	711/148
☐	5905488	May 1999	Demers et al.	345/173
☐	5974536	October 1999	Richardson	712/215
☐	6018793	January 2000	Rao	711/150
☐	6085029	July 2000	Kolawa et al.	395/183.14

OTHER PUBLICATIONS

Ian Foster, Designing and Building Parallel Programs, Addison-Wesley Publishing Company, 1995.
Barry Wilkinson and Michael Allen, Parallel Programming, Prentice-Hall, 1999.
David E. Culler and Jaswinder Pal Singh, Parallel Computer Architecture, Morgan Kaufman Publishers, Inc., 1999.
Sun Microsystems Computer Company, Prism 5.0 Reference Manual, Revision A., Nov. 1997.
"Purify for Windows NT, Product Overview," Ver. 6.0, available online at: www.rational.com/products/purify nt/index.jtmpl as of Jan. 11, 1999, 4 pages.
GeoMAMOS Project Home Page, available online at: www.ece.nwu.edu/.about.theory/geomamos.html as of Jun. 10, 1999, 4 pages.
"Introduction," available online at: www.ece.nwu.edu/.about.theory/gs tech 1 html/section3 1.html as of Jun. 10, 1999, 2 pages.
"Visualization for Developing Geometric Algorithms," available online at: www.ece.nwu.edu/.about.theory/gs_tech_1_html/section3 3.html as of Jun. 10, 1999, 2 pages.
"Algorithm Visualization System: Introduction," available online at: www.cp.eng.chula.ac.th/faculty/spi/research/avis/intro.html as of Jun. 10, 1999, 2 pages.
"Scientific Simulations and Algorithm Visualizations Using NESL and Java," available online at: www.cs.cmu/edu/.about.scandal/applets/ as of Jun. 10, 1999, 1 page.

ART-UNIT: 2122

PRIMARY-EXAMINER: Powell; Mark R.

ASSISTANT-EXAMINER: Zhen; Wei

ATTY-AGENT-FIRM: Finnegan, Henderson, Farabow, Garrett & Dunner, L.L.P.

ABSTRACT:

Methods, systems, and articles of manufacture consistent with the present invention assist a programmer in the optimization of an application program by displaying information about memory use in a manner useful to the programmer. A programmer selects an application program that he wishes to optimize. The application program is submitted to an instrumentation process that inserts additional instructions into the code of the application program so that, when executed, the instrumented code produces an output file, called a trace output file. The trace output file contains information about memory accesses. The trace output file is then submitted to a second program, or visualizer, that visually displays the memory accesses using a different visual effect for each type of memory access operation. The visualizer may be run at different speeds, forwards or backwards, paused, or may step through the memory accesses frame-by-frame. This visual representation of memory accesses allows programmers to recognize patterns of memory use that can be used to optimize the application program.

18 Claims, 8 Drawing figures
Exemplary Claim Number: 1
Number of Drawing Sheets: 8

BRIEF SUMMARY:

1 BACKGROUND OF THE INVENTION

2 A. Field of the Invention

3 This invention relates generally to methods for optimizing performance of computer programs and, more particularly, to methods for analyzing memory use in optimizing performance of computer programs.

4 B. Description of the Related Art

5 Computer programmers are continuously searching for ways to make application programs run faster and more efficiently. The performance time of an application may be minimized by reducing the time needed to execute each operation. Many application programs today, however, involve thousands of lines of code and millions of operations thereby making optimization of code a difficult process.

6 Almost each operation in an application program involves the memory of a computer in some capacity. Memory is "read" when an instruction in the application looks at (and generally uses) the contents of some area of memory without alteration. A "write" to memory takes place when an instruction of the program stores information in an area of memory, overwriting what, if anything, was previously stored in that memory location. Memory accesses may be either "reads" or "writes" to memory. Since a large number of operations of an application program involve accessing memory, performance may be improved by reducing the number of memory accesses performed by an application program.

7 One important step in optimizing an application program is to understand how the memory of the computer is allocated and how the memory is accessed during operation of the program. Conventionally, programmers study how the application works by, for example, inserting "print" instructions in the program that when executed, output the contents of memory at various points in the program operation, as process known as "instrumentation." This conventional method produces lengthy printouts of memory accesses that require close further analysis by a programmer and patterns of memory use are not easily detected. Programmers also frequently resort to a painstaking process of hand drawing representations of memory to aid the programmer in visually tracing memory accesses.

8 Multiple parallel processors may be used to improve the execution time of application programs. Application programs for a multiprocessor environment, however, are complicated to debug and optimize. To write new or modify existing application programs for use on multiple parallel processors, programmers must have a clear understanding of how each operation in a program interacts with memory and which specific memory elements are accessed. By determining where in a program the number of reads and writes is concentrated, programmers can identify portions of application code that need to be optimized. The execution time of these sections of program code with a high number of memory accesses may then be improved, for example, by replacing the section with optimized code or using a special purpose hardware device.

9 Another method for optimizing application programs is to identify and exploit data dependencies. Data dependencies arise when one computation requires the result of a previous computation. In multiplying two matrices, for example, items in the matrices are multiplied, then those results are summed. To arrive at the correct result, the sum must be performed after the multiplication. Program code that performs a matrix multiplication, will access some memory elements repeatedly. When a matrix multiplication is programmed for parallel processors, programmers must determine when certain memory elements are read for the last time, so that when the program writes to that matrix entry, no matrix information is lost.

10 There exist hardware and software devices that track accesses to particular memory elements. These devices, however, do not identify whether the particular access is a "read" or a "write." Furthermore, these conventional devices do not track data dependencies.

11 There exist also conventional tools that check for memory errors during execution of an application program. Purify.TM. from Pure Software Inc. and Insure++.TM. from ParaSoft Corporation are examples of tools that will perform run time memory checks for programs written in the C and C++ programming languages. These conventional tools help detect run-time memory errors arising from improper memory accesses, such as reading or writing beyond the bounds of an array, reading or writing to freed memory, freeing memory multiple times, reading and using uninitialised memory, and forgetting to free memory (called

"memory leaks") once a program has terminated. These tools, however, produce lengthy printouts that need further analysis by a programmer. Furthermore, conventional tools, like Purify and Insure++, notify the programmer only when these error conditions occur. Additionally, conventional memory-usage analysis tools do not use visual displays to track use of individual memory areas.

12 Conventional tools used to debug application programs may visually display regions of memory using various colors depending on the value in the memory element. Debugging tools are used to identify and correct errors that occur during execution of an application program. These conventional visual debugging tools are useful for analyzing how values in memory change as an application program executes, but do not show when memory is accessed but not changed. For example, it does not show "reads" because during reads the value stored in memory is not changed. Furthermore, debugging tools typically do not show the frequency that a particular memory element is accessed.

13 Some compiler optimizers do data flow analysis when compiling application programs. Optimization by compiler optimizers, however, is performed at a basic block level and not generally performed for an entire program. Furthermore, compiler optimizers guess at data dependencies merely by looking at the source code of a program, not by running the code. Some data dependencies, however, are not detectable until runtime. Some sophisticated models may make good approximations, but these are still approximations versus actual accesses.

14 Furthermore, some data dependencies are not always obvious to a compiler that looks only at the source code. For example, in the expression A(I)=A(I)+A(I+1), the data dependency between the last two terms will be recognized by many conventional compilers. However, the following example shows a set of instructions with a data dependency that would not be obvious to most conventional compilers.

15 Do J=1,10

16 I=B(j)

17 A(I)=A(I)+A((B(J+1))

18 Enddo

19 This program code adds the i.sup.th element of array A to another array element, which is determined by B(J+1). If B is an array of independent, unrelated values, the relationship between the last two terms is not easily determined at the time when the instructions are compiled.

20 Therefore, a need exists for an improved method of optimizing programs by analyzing memory accesses. Furthermore, a need exists for a method of analyzing memory accesses that identifies data dependencies and displaying this information in a manner useful for a programmer.

21 SUMMARY OF THE INVENTION

22 Systems, methods, and articles of manufacture consistent with the present invention track accesses to memory during execution of an application program. Additional instructions are inserted into the application program and the instrumented application program is executed. During execution of the instrumented application program, information reflecting different types of memory access operations is generated. This generated information is displayed in a visual form reflecting the different types of memory access operations using a different visual effect for each type of memory access operation.

DRAWING DESCRIPTION:

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, which are incorporated in and constitute a part of this specification, illustrate an embodiment of the invention and, together with the description, serve to explain the advantages and principles of the invention. In the drawings,

FIG. 1 shows a block diagram of an exemplary system in which methods consistent with the present invention may be implemented;

FIG. 2 shows a block diagram of an exemplary multiprocessor computer system in which application programs written in a manner consistent with the present invention may be implemented;

FIG. 3 is a flow diagram representing operations of a method consistent with the present invention;

FIG. 4 is an example of an application program;

FIG. 5 is an example of an instrumented application program based on the application program of FIG. 4, consistent with the present invention;

FIG. 6 is an example of the output generated by the modifications included in the program of FIG. 5, consistent with the present invention;

FIG. 7 is an example of a graphical display showing the a trace file of a matrix factorization program consistent with the present invention; and

FIG. 8 is an example of a graphical display using a histogram.

DETAILED DESCRIPTION:

1 DETAILED DESCRIPTION

2 Reference will now be made in detail to an implementation consistent with the principles of the present invention as illustrated in the accompanying drawings. Wherever possible, the same reference numbers will be used throughout the drawings and the following description to refer to the same or like parts.

3 A. Introduction

4 Methods, systems, and articles of manufacture consistent with the present invention assist a programmer in the optimization of an application program by displaying information about memory use in a manner useful to the programmer. A programmer selects an application program that he wishes to optimize. The application program is submitted to an instrumentation process that inserts additional instructions into the code of the application program so that, when executed, the instrumented code produces an output file, called a trace file. The trace file contains information about memory accesses. The trace file is then submitted to a second program, which may be referred to as a visualizer, that visually displays the memory accesses using a different visual effect for each type of memory access operation. The visualizer may be run at different speeds, forwards or backwards, paused, or may step through the memory accesses frame-by-frame. This visual representation of memory accesses allows programmers to recognize patterns of memory use that can be used to optimize the application program.

5 B. Systems and Apparatus

6 FIG. 1 is a block diagram that illustrates a data processing system 100 upon which embodiments of the invention may be implemented. System 100 includes a bus 102 or other communication mechanism for communicating information, and a processor 104 coupled with bus 102 for processing information. System 100 also includes a memory 106, which can be a random access memory (RAM) or other dynamic storage device, coupled to bus 102 for storing information, such as

memory access information or instructions to be executed by processor 104. Memory 106 also may be used for storing temporary variables or other intermediate information during execution of instructions to be executed by processor 104. System 100 further includes a read only memory (ROM) 108 or other static storage device coupled to bus 102 for storing static information and instructions for processor 104. A storage device 110, such as a magnetic disk or optical disk, is provided and coupled to bus 102 for storing information and instructions.

7 System 100 may be coupled via bus 102 to a display 112, such as a cathode ray tube (CRT) or liquid crystal display (LCD), for displaying information to a computer user. An input device 114, including alphanumeric and other keys, is coupled to bus 102 for communicating information and command selections to processor 104. Another type of user input device is cursor control 116, such as a mouse, a trackball or cursor direction keys for communicating direction information and command selections to processor 104 and for controlling cursor movement on display 112. This input device typically has two degrees of freedom in two axes, a first axis (e.g., x) and a second axis (e.g., y), that allows the device to specify positions in a plane.

8 In accordance with the principles of the present invention, system 100 may be used for analyzing accesses to memory. Consistent with one implementation of the invention, information from the multiple remote resources, such as memory access information, is provided by system 100 in response to processor 104 executing one or more sequences of one or more instructions contained in memory 106. Such instructions may be read into memory 106 from another computer-readable medium, such as storage device 110. Execution of the sequences of instructions contained in memory 106 causes processor 104 to perform the process states described herein. In an alternative implementation, hard-wired circuitry may be used in place of or in combination with software instructions to implement the invention. Thus implementations of the invention are not limited to any specific combination of hardware circuitry and software.

9 The term "computer-readable medium" as used herein refers to any media that participates in providing instructions to processor 104 for execution. Such a medium may take many forms, including but not limited to, non-volatile media, volatile media, and transmission media. Non-volatile media includes, for example, optical or magnetic disks, such as storage device 110. Volatile media includes dynamic memory, such as memory 106. Transmission media includes coaxial cables, copper wire, and fiber optics, including the wires that comprise bus 102. Transmission media can also take the form of acoustic or light waves, such as those generated during radio-wave and infra-red data communications.

10 Common forms of computer-readable media include, for example, a floppy disk, a flexible disk, hard disk, magnetic tape, or any other magnetic medium, a CD-ROM, any other optical medium, punch cards, papertape, any other physical medium with patterns of holes, a RAM, PROM, and EPROM, a FLASH-EPROM, any other memory chip or cartridge, a carrier wave, or any other medium from which a computer can read.

11 Various forms of computer readable media may be involved in carrying one or more sequences of one or more instructions to processor 104 for execution. For example, the instructions may initially be carried on magnetic disk of a remote computer. The remote computer can load the instructions into its dynamic memory and send the instructions over a telephone line using a modem. A modem local to computer system 100 can receive the data on the telephone line and use an infra-red transmitter to convert the data to an infra-red signal. An infra-red detector coupled to bus 102 can receive the data carried in the infra-red signal and place the data on bus 102. Bus 102 carries the data to memory 106, from which processor 104 retrieves and executes the instructions. The instructions received by memory 106 may optionally be stored on storage device 110 either before or after execution by processor 104.

12 System 100 also includes a communication interface 118 coupled to bus 102.

Communication interface 118 provides a two-way data communication coupling to a network link 120 that is connected to local network 122. For example, communication interface 118 may be an integrated services digital network (ISDN) card, a cable modem, or a modem to provide a data communication connection to a corresponding type of telephone line. As another example, communication interface 118 may be a local area network (LAN) card provide a data communication connection to a compatible LAN. Wireless links may also be implemented. In any such implementation, communication interface 118 sends and receives electrical, electromagnetic or optical signals that carry digital data streams representing various types of information.

13 Network link 120 typically provides data communication through one or more networks to other data devices. For example, network link 120 may provide a connection through local network 122 to a host computer 124 and/or to data equipment operated by an Internet Service Provider (ISP) 126. ISP 126 in turn provides data communication services through the Internet 128. Local network 122 and Internet 128 both use electric, electromagnetic, or optical signals that carry digital data streams. The signals through the various networks and the signals on network link 120 and through communication interface 118, which carry the digital data to and from computer system 100, are exemplary forms of carrier waves transporting the information.

14 System 100 can send messages and receive data, including program code, through the network(s), network link 120 and communication interface 118. In the Internet example, a server 130 might transmit a requested code for an application program through Internet 128, ISP 126, local network 122 and communication interface 118. In accordance with the present invention, one such downloaded application allows a user to select security countermeasures and countermeasure strength levels, as described herein. The received code may be executed by processor 104 as it is received, and/or stored in storage device 110, or other non-volatile storage for later execution. In this manner, system 100 may obtain application code in the form of a carrier wave.

15 Although system 100 is shown in FIG. 1 as being connectable to one server, 130, those skilled in the art will recognize that system 100 may establish connections to multiple servers on Internet 128. Additionally, it is possible to implement methods consistent with the principles of the present invention on other device comprising at least a processor, memory, and a display, such as a personal digital assistant.

16 It is further possible to implement methods consistent with the principles consistent with the present invention in a multiprocessor environment. There are many configurations for such a multiprocessor computer system, one of which is illustrated in FIG. 2. For example, in a tightly-coupled configuration, the multiple processors of a system may all be located in the same physical box. In an alternative, loosely-coupled arrangement the system may be formed by multiple computers in a network, each computer having a separate processor.

17 FIG. 2 is a block diagram of multiprocessor computer system 200 with which methods consistent with the present invention may be implemented. Multiprocessor computer system 200 comprises a single, shared memory 230 and multiple processors 240a, 240b, . . . 240n. The number and type of processors is not critical to execution of the data flow program developed in accordance with the present invention. For example, an HPC Server with a multiple processor configuration may be used. The HPC Server is a product of Sun Microsystems, Inc. Processors 230.sub.a through 230.sub.n represent processors that communicated with shared memory 210. Processors 230a through 230n may be, for example, stand-alone computers, a collection of processors contained on a either a computer or processor board, or a combination. Each of processors 230a through 230n may be, for example, a system 100 as described above and shown in FIG. 1. As shown in FIG. 2, multiprocessor computer system 200 is connected to a network interface 820 for transmitting and receiving data, including memory access information, across a network.

18 Shared memory 230 may correspond to a portion of memory designated for use by

several processors, or alternatively it may represent a local cache of a single processor that cooperates with other processors. This shared memory may be either a virtual or a global region of main memory.

19 C. Process

20 FIG. 3 is a flow chart illustrating the operations of a process for analyzing memory use consistent with the present invention. Consistent with the present invention, the method begins by inputting or obtaining an application program to be optimized (step 310). The application program may be input to the computer from a disk via a disk drive or received over a network connection. The application program may also be obtained from storage resident on the computer. FIG. 4 shows an example of an application program that computes an average of five elements of matrix A and stores the result in an element of matrix B (lines 432-456). In the second subroutine, matrix B is copied to matrix A (lines 460-476).

21 Next, memory regions of interest are selected (step 320). A memory region may be a specific memory address or a set of memory addresses. Memory regions may be selected, for example, by a user manipulating a user interface to indicate a subset of memory regions from a set of available memory regions. The method may also proceed without selection by operating on all available memory regions.

22 In step 330, the program is instrumented to include code for creating an output file. Instrumentation involves adding statements and commands to the source code which, when executed, will produce an output file. In one example, the commands call a set of routines in, for example, a library, and they are used to record the memory information. The library may contain, for example, functions for initializing the environment, creating objects, logging events, and registering states. Initialization statements and declarations, for example, may be added to the source code to set up the data structures needed to initialize the library so that it can accept access entries. The majority of statements added to the source code will be Log Update type entries, since every memory access for a monitored region that was selected in step 320 requires a call to the library to record the event. Instrumented source code is output to a file (step 340).

23 FIG. 5 shows the application program of FIG. 4 after instrumentation. Lines 512 through 520 have been added to initialize variables that will be used later in the instrumented program. Lines 522 through 530 call routines that perform additional initializations. Lines 536 through 550 and lines 564 through 570 have been added to call routines that log information about memory accesses during the two loops.

24 After the application sources files have been instrumented, the instrumented source code is compiled (step 350) and run normally (step 360). Execution of the instrumented source code will produce a trace output file that describes the memory accesses that occur during execution of the application program (step 370). An example of a trace output file consistent with the present invention is shown in FIG. 6. Line 602 of the trace output file states: "CREATE A 159288 2 4 4 4" and indicates that a monitored memory region is being created and it is named "A." Memory region A starts at memory address 159288 and is a two-(2) dimensional region. Each two-dimensional region has a size of four (4) bytes, and the region is made up of four (4) elements along the x-axis and four (4) elements along the y-axis. This information allows the program to track the region that is being accessed simply by the address in memory. An additional side benefit of memory access tracking in this manner is that out-of-bounds accesses are immediately discovered and reported.

25 Lines 606 and 608 of the trace output file initialize two states of the algorithm called "Pt Stencil" and "Matrix Copy." On line 606, the state "Pt Stencil" is assigned the number 33. On line 608, the state "Matrix Copy" is assigned the number 34. Lines 610 through 684 in the trace output file are a listing of the memory accesses and other memory access information. Column 1, for example, contains information describing the size of the memory area

accessed. In FIG. 6, the entry "RW" indicates a single access to one memory region. Other indicators may indicate, for example, multiple accesses to a regional of memory, such as to a vector, matrix, or three-dimensional volume of memory areas.

26 Column 2 in FIG. 6 contains an indicator corresponding to whether the access was a read (0) or a write (1). This column may also indicate, for example, last write (2), last read (3) or other such information. Column 3 shows the starting address of the memory that was accessed. As shown on line 610 of FIG. 6, for example, the first memory address accessed is memory address "159304".

27 The fourth column is an integer referring to the state from which the current access originated. During the instrumentation process, the first loop was assigned the name "Pt Stencil" and the value 33. As shown in lines 610 through 630, the last column indicates the memory access described on this line was executed during state 33, or during "Pt Stencil."

28 In step 380, the outputted trace file is post-processed to display the information contained in the trace file visually on a display device. In post-processing, the access trace file may be, for example, fed into a visualizer that conditions the data for display. Since the memory access information is stored in a trace output file, instead of being displayed in real-time as the instrumented application executed, display of the information is more flexible and more adaptable to the programmer's needs. The visualizer, for example, may display the contents of the trace output file forward, backward, faster, slower, by time step, or by state transition.

29 The results of post-processing are displayed visually (step 390). FIG. 7 is an example of eight snapshots of a visual display consistent with the present invention. The diagrams in FIG. 7 display exemplary memory region accesses that occur during various stages of a matrix factorization algorithm. Each of the matrices represents a different memory region. The memory region on the left may, for example, represent the elements of matrix A; the memory region on the right represents the elements of matrix B. Each set of two matrices represents the memory regions during different stages of the algorithm. As indicated by the key, regions that are being read are displayed using one visual means. A visual means may be a pattern, shape, color or combination thereof. Memories elements that are being written to are displayed using a second visual means. Each time one of the interested memory regions is read or written to, the visual means of display for that element changes. For example, if different colors are used to represent difference memory access types, a read may the entry red and a write may then turn the block green. A third visual means, such as the solid (completely filled) blocks in FIG. 7, may indicate that a memory element was written to and read in the same step.

30 The trace output file may also be used to display, for example, the last time that the algorithm wrote to a particular element. The visualizer program will, for example, examine the trace output file from the end and code the last write to be displayed in a different pattern or color (i.e. light blue). The "last write" information is particularly useful in developing software for parallel processors. By examining the display shown in FIG. 7, for example, the programmer will recognize that a lot of computations are done in the first column of the "B" matrix, so these calculations may be optimized using parallel processing. If so, it is important to know when the last write occurs for each memory element, because it indicates that the contents may then be made available for the next calculation.

31 The trace output file may also be used to display the frequency of accesses to certain areas of memory. FIG. 8, for example, shows access frequencies displayed using a histogram. Color-coded areas represents frequent accesses to memory and identifies information that should remain in cache. Frequency access information may also be displayed, for example, as color-coded bumps on a line or bars with height on a plane giving a three-dimensional representation of the memory. Also consistent with the principles of the present invention is a display that shows data dependencies as links between each of the dependent

blocks and the blocks from which they depend. A block is dependent on another block when the first block requires the result of the second block for the first block to perform properly within the system. Yet another display could use a variety of colors within a three-dimensional cube that could be rotated.

32 The present invention may also be used to display and analyze the memory elements of multiple processors in a multiprocessor environment. In a system with multiple processors, each processor executes one or more threads, depending upon the number of processors, to achieve multi-processing of the program. Multithreading is the partitioning of a computer program or application into logically independent "threads" of control that can execute in parallel. Each thread includes a sequence of instructions and data used by the instructions to carry out a particular program task, such as a computation or input/output function. Methods and systems for coordinating the distribution of shared memory to threads of control executing in a parallel computing environment are described in related U.S. patent application Ser. No. 09/244,135, to Shaun Dennie, filed on Feb. 9, 1999, entitled "Protocol for Coordinating the Distribution of Shared Memory," the contents of which are hereby incorporated by reference.

33 In a multiprocessor parallel environment, such as the one described in Dennie, each thread may be assigned a specific region of a shared memory. To visually display accesses to the entire shared memory in time-sequential order, the trace output file in a multiprocessor environment also must contain information regarding the time of each access.

34 D. Conclusion

35 Methods, systems, and articles of manufacture consistent with the present invention thus provide tools for analyzing memory use. By displaying various types of memory accesses graphically, programmers can better understand where large numbers of operations are taking place and then optimize the corresponding sections of the application code to reduce the number of time-consuming memory accesses. By using an outputted trace file and a visualizer program, the programmer can display the memory accesses at various rates, such as forward, backwards, faster, slower, or step-by-step. By displaying information regarding memory use, such as "last read" and "last write," programmers are assisted in developing and optimizing application programs for multiprocessor environments.

36 Also, methods consistent with the present invention are applicable to all programs for execution in a multiprocessor system regardless of the computer programming language. For example, Fortran 77 is a programming language commonly used to develop programs for execution by multiprocessor computer systems.

37 The foregoing description of an implementation of the invention has been presented for purposes of illustration and description. It is not exhaustive and does not limit the invention to the precise form disclosed. Modifications and variations are possible in light of the above teachings or may be acquired from practicing of the invention. For example, the described implementation includes software but the present invention may be implemented as a combination of hardware and software or in hardware alone. The invention may be implemented with both object-oriented and non-object-oriented programming systems. The scope of the invention is defined by the claims and their equivalents.

CLAIMS:

What is claimed is:

1. A computer-implemented method of analyzing accesses to a memory by an instrumented application program, comprising:

executing the instrumented application program;

generating, during the execution of the instrumented application program, information reflecting different types of memory access operations; and

displaying the generated information in a visual form reflecting the different types of memory access operations using a different visual effect for each type of memory access operation.

2. The method of claim 1, wherein displaying the generated information further comprises:

displaying an array, wherein each cell of the array represents a memory region; and

displaying a first cell using a first visual means when the memory region corresponding to the first cell is read.

3. The method of claim 1, wherein displaying the generated information further comprises:

displaying an array, wherein each cell of the array represents a memory region; and

displaying a second cell using a second visual means when information is written to the memory region corresponding to the second cell.

4. The method of claim 1, wherein displaying the generated information further comprises:

displaying an array, wherein each cell of the array represents a memory region; and

displaying a third cell using a third visual means when information is written to the memory region corresponding to the third cell for the last time in execution of the application.

5. The method of claim 1, wherein displaying the generated information further comprises:

displaying an array, wherein each cell of the array represents a memory region; and

displaying a fourth cell using a fourth visual means when information is read from the memory region corresponding to the fourth cell for the last time in execution of the application.

6. The method of claim 1, wherein displaying the generated information further comprises:

displaying an array, wherein each cell of the array represents a memory region; and

displaying a fifth cell using a fifth visual means when information is read to and written from the memory region corresponding to the fifth cell in the same operation of the application.

7. An apparatus for tracking accesses to memory in an application, the apparatus comprising:

a memory having program instructions, and

a processor configured to use the program instructions to:

execute the instrumented application program;

generate, during the execution of the instrumented application program,

information reflecting different types of memory access operations; and

display the generated information in a visual form reflecting the different types of memory access operations using a different visual effect for each type of memory access operation.

8. The apparatus of claim 7, wherein the program instruction to display the information includes instructions to:

display an array, wherein each cell of the array represents a memory region; and

display a first cell using a first visual means when the memory region corresponding to the first cell is read.

9. The apparatus of claim 7, wherein the program instruction to display the information includes an instruction to:

display an array, wherein each cell of the array represents a memory region; and

display a second cell using a second visual means when information is written to the memory region corresponding to the second cell.

10. The apparatus of claim 7, wherein the program instruction to display the information included an instruction to:

display an array, wherein each cell of the array represents a memory region; and

display a third cell using a third visual means when information is written to the memory region corresponding to the third cell for the last time in execution of the application.

11. The apparatus of claim 7, wherein the program instruction to display the information includes an instruction to:

display an array, wherein each cell of the array represents a memory region; and

display a fourth cell using a fourth visual means when information is read from the memory region corresponding to the fourth cell for the last time in execution of the application.

12. The apparatus of claim 7, wherein the program instruction to display the information includes an instruction to:

display an array, wherein each cell of the array represents a memory region; and

display a fifth cell using a fifth visual means when information is read to and written from the memory region corresponding to the fifth cell in the same operation of the application.

13. A computer-readable medium containing instructions for controlling a computer system to perform a method, the computer system having a group of data structures reflecting a logical structure of a data source, the method comprising:

executing the instrumented application program;

generating, during the execution of the instrumented application program, information reflecting different types of memory access operations; and

displaying the generated information in a visual form reflecting the different

types of memory access operations using a different visual effect for each type of memory access operation.

14. The computer-readable medium of claim 13, wherein displaying the generated information further comprises:

displaying an array, wherein each cell of the array represents a memory region; and

displaying a first cell using a first visual means when the memory region corresponding to the first cell is read.

15. The computer-readable medium of claim 13, wherein displaying the generated information further comprises:

displaying an array, wherein each cell of the array represents a memory region; and

displaying a second cell using a second visual means when information is written to the memory region corresponding to the second cell.

16. The computer-readable medium of claim 13, further including:

displaying an array, wherein each cell of the array represents a memory region; and

displaying a third cell using a third visual means when information is written to the memory region corresponding to the third cell for the last time in execution of the application.

17. The computer-readable medium of claim 13, further including:

displaying an array, wherein each cell of the array represents a memory region; and

displaying a fourth cell using a fourth visual means when information is read from the memory region corresponding to the fourth cell for the last time in execution of the application.

18. The computer-readable medium of claim 13, further including:

displaying an array, wherein each cell of the array represents a memory region; and

displaying a fifth cell using a fifth visual means when information is read to and written from the memory region corresponding to the fifth cell in the same operation of the application.

US-PAT-NO: 5524244

DOCUMENT-IDENTIFIER: US 5524244 A

TITLE: System for dividing processing tasks into signal processor and decision-making microprocessor interfacing therewith

---------- KWIC ---------

US Patent No. - PN (1):
5524244

Parent Case Text - PCTX (2):
This application is a continuation of application Ser. No. 07/776,161 filed on Oct. 15, 1991, and issued as U.S. Pat. No. 5,287,511 on Feb. 15, 1994, the complete disclosure of which is incorporated herein by reference.

Brief Summary Text - BSTX (65):
Based on the above break-down of tasks it can be seen that SCDSPs are called upon to do both of what may be termed "signal processing" and "logic processing". Signal processing is typically computationally intensive, requires low latency and low parasitic overhead for real time I/O, must efficiently execute multiple asynchronous deterministic processes, and be controllable. Real time signal processors are typically controllable processors which have very large I/O bandwidths, are required to conduct many millions of computations per second, and can conduct several processing functions in parallel. In contrast to signal processing, logic processing is

usually memory intensive (as opposed to computationally intensive), must
efficiently handle multiple interrupts (particularly in a multiprocessor
system), and acts as a controller (as opposed to being controllable). A common
type of logic processor is the microprocessor which relies on extensive
decision oriented software to conduct its processes. This software is
typically written in a high level language such as "C". The code often
contains numerous "if . . . then . . . else" like constructs which can result
in highly variable execution times which are readily dealt with in non-real
time applications, but present highly problematical scheduling problems for
efficient real time systems.

Detailed Description Text - DETX (29):

The multiplier block 430 and the ALU block 450 of the GSP perform the
numerical computations for the GSP. The multiplier block 430 is comprised of
two input registers Xreg 432 and Yreg 434, a multiplexer 436 which is coupled
to the internal bus 490 via tristate driver 449a, a multiplier 438 with a post
Xreg 439, and a multiplier control 441, a summer 442, an output register Mreg
444, and a second multiplexer 446 which selects which of six words is to be
output onto internal bus 490 via tristate driver 449b. Typically, the
multiplicand is loaded into Xreg 432. Then the multiplier is loaded into Yreg
434 while the multiplicand is loaded into post Xreg 439. The multiplier
control 441 permits the multiplier 438 to function over several machine clock
cycles (e.g. three clock cycles totaling 300 nanoseconds=fifteen internal GSP
cycles). If in multiplying, the multiplier overflows, a status flag M is set,
and this information is conveyed to the branch logic block 432 of the program

logic section 420. Regardless, the product of the multiplier and multipicand
is forwarded to summer 442 which, in a multiply with accumulate mode, adds the
new product to the sum of previous products and forwards the sum to the
multiply register M 444. In a pure multiply mode, the contents of the summer
are cleared so that the product is forwarded through the summer which adds zero
and send the product to the M register.

Detailed Description Text - DETX (32):
Wait flag block 460 is comprised of two wait flag registers WFreg 462 and
DFreg 464, a multiplexer 466, and OR gate 468. The bits of the wait flag
registers may be set (i.e. written to) by data sent over the internal bus 490.
Also, registers WFreg 462 and DFreg 464 are coupled to a flag bus 198 which is
written to each time predetermined locations in the data RAM 125 are addressed
as hereinbefore described with reference to FIGS. 2 and 13. In this manner,
each bit of the wait flag registers 462 and 464 may be selectively cleared.
When all of the bits in register WFreg 462 have been cleared due to the
occurrences of specified events (e.g. the data RAM has received all the
information which is required for another computation), OR gate 468 is used to
provide a status flag W which indicates the same. Status flag W is read by the
branch logic block 432. In this manner, "jump on wait flag" commands may be
executed.

Detailed Description Text - DETX (50):
When the value of counter 632 is incremented to the value stored in the
index length register 622, the last location in the FIFO has been addressed.
Accordingly, it is desirable to recirculate; i.e. to continue by addressing the

first location in the FIFO. With the value of counter 632 equal to the value
of register 622, bus wide XNOR gate 645 provides a high signal which is passed
through three input OR gate 646. As a result, counters 630, 632, and 634 are
reset. As indicated in FIG. 4a, a "clear counter" signal may also be generated
by a power up reset (PUR) signal which is generated by applying a signal to a
predetermined pin (not shown) of the SPROC, and by a SYNC signal which is
generated by writing to address 0405H of the data RAM 100. The SYNC signal
permits different DFMs to be synchronized to each other.

Detailed Description Text - DETX (51):

If desired, the input section of one DFM can be synchronized to the output
section of the same or another DFM. This synchronization is accomplished via a
pin (not shown) on the SPROC which generates the "en buf" input into OR gate
644. In turn, OR gate 644 provides a high signal which resets counter 634 in
synchronization with the resetting of a similar counter in a DFM output section
such as described with reference to FIG. 4b.

Detailed Description Text - DETX (58):

The remaining blocks of the DFM output section include a FIFO length
register 680, a buffer length register 682, a sample counter 684, a divide by
two block 685, comparators 686 and 687, a bus wide OR gate 689, and a set/reset
block 690. The FIFO length register 682 stores the full length of the FIFO.
When the value of the offset counter 656 is equal to the FIFO length stored in
buffer 680, a sync pulse is generated by bus wide XNOR gate 686 which is used
to synchronize the incoming data signal into an input section of a DFM with the
outgoing data signal from the described output DFM. The sync pulse generated

is received by the input section of the DFM (seen in FIG. 4a) as the signal enbuf1, previously described. In addition the sync pulse may be used to reinitialize the DFM by clearing the offset counter 656 and reloading the registers. When the value in the offset counter 656 is equal to one-half the value of the FIFO length register 680 (as determined by divide by two block 685), comparator 687 provides a pulse to set/reset block 690 which is indicative of the fact that the address placed on the data RAM bus is the address half-way through the data RAM buffer associated with the particular DFM. When the data RAM address is the half-full address, the data being written into the data RAM buffer should not be written into the half-full address (i.e. there should never exist a situation where the address is being written to and read from at the same time). Thus, if D type flip-flop 667 provides a high signal to AND gate 670 while the DFM is running, and the output from set/reset block 690 is also, high, AND gate 698 provides a high output which sets an error flag for the DFM.

Detailed Description Text - DETX (61):

Turning to FIG. 5a, a block diagram of the serial input port 700a of the invention is seen. The basic function of the serial input port is to receive any of many forms of serial data and to convert the received serial data into parallel data synchronous with the internals of the SPROC and suitable for receipt by the DFM 600 and for transfer onto the data RAM bus 125. To accomplish the basic function, the serial input port has a logic block 710, a data accumulation register 720, and a latched buffer 730. The logic block 710 and the data register 720 are governed by seven bits of information programmed

into the serial input port 700a upon configuration during boot-up of the SPROC
10. The seven bits are defined as follows:

Detailed Description Text - DETX (63):
Once the data is properly accumulated in register 720, it is latched into
buffer 730 where it is held until it can be forwarded through the input section
of the DFM 600 for storage in the multiported RAM 100. The holding of the data
in the buffer 730 until the appropriate signal is received effectively causes
data which is asynchronous with the SPROC 10 to become synchronized within the
SPROC system.

Detailed Description Text - DETX (69):
Regardless of how the data input register 812 is filled, after the data is
assembled, the host port 800 awaits an enabling signal from the SPROC timing so
that it can write its twenty-four bit word to the data RAM bus 125 via driver
817 or the program RAM bus 155 via driver 815. In this manner, the host port
800 synchronizes data to the SPROC 10 which was received in a manner
asynchronous to the SPROC 10. The address to which the data is written is
obtained from the twelve bit address section A0-A11 of the host bus 165. The
twelve bit address is forwarded from host bus 165 to the address input register
820. When the host port 800 is enabled, if the address contained in the
address input register 820 is indicative of a data RAM location, the address is
placed via driver 822 on the sixteen bit address section of the data RAM bus
125. Because the address bus is a sixteen bit bus, while the address in
address input register 820 is a twelve bit address, four zeros are added as the
msbs of the address via driver 824 when the address and data are put on the

data RAM bus. If the address contained in the address input register 820 is
indicative of a program RAM location (address location 1K and below), the
address is placed via driver 826 on the twelve bit address section of the
program RAM bus 155.

Detailed Description Text - DETX (161):
Several cells, including those used for microprocessor access, are described
in detail below with reference to function, algorithm, terminals, parameters,
macro keys, execution rime, resource usage, and icon. The function provides a
brief description of the operations or calculations performed by the cell. The
algorithm (where applicable) details the methodology used to implement the cell
function. Terminals are the inputs and outputs for a cell. Each terminal is
associated with a pin number on the cell's icon. The variable type, range of
legal values, and default value are provided for each terminal. Parameters are
specifications that define the function of a particular instance of a cell.
Parameter names and default values (where applicable) are provided for each
cell. Parameter descriptions use the exclusive OR character (1) in listings of
legal parameter values. This character indicates that only one of the listed
choices may be used. Execution time is the maximum number of instruction
cycles required to complete the code for a cell instance. Execution time
differs for the in-line form and subroutine form (where applicable) of each
cell. Resource usage is the number of memory locations required by the cell.
Resources include program memory allocations for instructions and data memory
allocations for variables. Resource usage differs for the in-line form and
subroutine form (where applicable) of each cell. Each cell is represented in

the graphical display as an icon. Other examples of cell icons can be seen in
FIG. 11 discussed in detail below. Source code for several of the cells
described below is attached to disclosure of U.S. Pat. No. 5,287,511 as
appendix B and incorporated herein by reference.

Detailed Description Text - DETX (202):
The frequency-domain properties of a filter may be evaluated for any
wordlength. Computation of a frequency response using quantized coefficients
serves to illustrate the degree of sensitivity of the filter performance to the
use of finite precision coefficients, i.e., the degree to which the poles of
IIR filters, and the zeros of IIR and FIR filters, are modified by the
finite-precision coefficient values.

Detailed Description Text - DETX (203):
The following limitations apply to filters designed using the filter design
interface: Maximum order for IIR filters is 20. Maximum length for PMR
(Equiripple) FIR filters is 200. Maximum length for Kaiser window FIR filters
is 511. Frequency response for IIR and FIR filters is limited to up to 500
spectrum values covering any desired segment of the frequency range between d-c
and one-half of the sampling frequency. The response computation may be
specified either by the number of points in a frequency range or by the spacing
between points on the frequency axis.

Detailed Description Text - DETX (214):
It is not unusual to approach the design of a filter without specific values
for all of the critical frequencies, having only a general idea of passband and
stopband locations. To aid in the design process, the filter design interface

provides full capability for adjusting all parameters of the filter to achieve
a best compromise between performance and complexity (as measured by filter
order). The procedure is fully interactive; all computations are done by the
filter design interface.

Detailed Description Text - DETX (232):
Realization of high performance filters--by which is usually meant sharp
cutoff--is restricted in the analog domain by component precision and
tolerance. For digital filters, precision refers to the wordlength of the
computation used in implementing the filter. There is no direct counterpart to
component tolerance; clock stability is a possible analogy. The SPROC chip
uses a 24-bit computational word, which is equivalent to a resolution of better
than 1 part in 106. The development system's crystal controlled clock provides
superior stability. All of this gives digital filters on the SPROC chip a
performance level that is far better than any analog implementation. Because
this high performance is so seemingly easy to achieve, the designer is often
seduced into overspecifying filter performance, with the penalty being
increased computational load and increased memory usage. In some cases there
will be additional signal-to-noise ratio degradation due to an accumulation of
quantization noise originating in the arithmetic rounding process; this effect
is significant only for IIR filters, because it is their inherent feedback
operation which can lead to an amplification of quantization noise.

Detailed Description Text - DETX (260):
2. The Schedule module takes the files produced by MakeSDL and adds the
code blocks for the cells used in the design (from the

function library or
user-defined cells) and any data files required in addition to those included
in the partial code package obtained from MakeSDL. Then the Schedule module
schedules the code according to on-chip resource availability and adds special
"glue" cells called phantoms that provide control and synchronization functions
for the general signal processors (GSPs) on the chip. These cells are included
in the SPROCcells function library, but reserved for internal use. The
Schedule module produces binary program and data files for the design. It also
produces a file of symbolic references to chip memory locations.

Detailed Description Text - DETX (314):
Turning to FIG. 10, a flow diagram of the SPROC and microprocessor
development environment is seen. At 2010, using graphic entry packages such as
"Draft", "Annotate", "ERC" and "Netlist" which are available from OrCad in
conjunction with cell library icons such are provided from a cell library 2015,
a block diagram such as FIG. 11 is produced by the user to represent a desired
system to be implemented. The OrCad programs permit the user to draw boxes,
describe instance names (e.g., multiplier 1, multiplier 2, etc. such as seen in
FIG. 11 as MULT1, MULT2, . . .), describe parameters of the boxes (e.g.,
spec=filter 1; or upper limit=1.9, lower limit=-1.9 such as seen in FIG. 11)
and provide topology (line) connections. The output of the OrCad programs is a
netlist (a text file which describes the instantiation, interconnnect and
parameterization of the blocks) which is fed to a program MakeSDL 2020 which
converts or translates the netlist output from OrCad into a netlist format more
suitable and appropriate for the scheduling and programming of the SPROC.

Source code for MakeSDL is attached to the disclosure of U.S. Pat. No. 5,287,511 as Appendix A. It will be appreciated that a program such as MakeSDL is not required, and that the netlist obtained from the OrCad programs (or any other schematic package program) can be used directly.

Detailed Description Text - DETX (317):

Both the netlist and data files output by the MakeSDL program are input to a scheduling/compiling program as indicated at 2040. In addition, a cell library 2015 containing other SDL files are provided to enable the scheduler/compiler to generate desired code. Among the signal processing functions provided in the cell library are a multiplier a summing junction, an amplifier, an integrator, a phase locked loop, an IIR filter, a FIR filter, an FFF, rectifiers, comparators, limiters, oscillators, waveform generators, etc. Details of the scheduler/compiler are described in more detail hereinafter, and source code for the scheduler/compiler is attached to the disclosure of U.S. Pat. No. 5,287,511 as Appendix M.

Detailed Description Text - DETX (319):

As aforementioned, the scheduler/compiler produces a symbol file (.sps) for use by the microprocessor. Depending upon the type of microprocessor which will act as a host for the SPROC, the symbol file will be translated into appropriate file formats. Thus, as shown in FIG. 10, symbol translation is accomplished at 2050. Source code in accord with the preferred embodiment of the invention is provided in Appendix C which is attached to the disclosure of U.S. Pat. No. 5,287,511, for a symbol translator which translates the .sps file generated by the scheduler/compiler 2040 to files which can be compiled

for use by a Motorola 68000 microprocessor. In accord with the preferred
embodiment, the symbol translator 2050 generates to files: a .c (code) file,
and a .h (header) file. The code file contains functions which can be called
by a C program language application. The header file contains prototypes and
symbol definitions for the microprocessor compiler hereinafter described.

Detailed Description Text - DETX (327):
Code for signal processing functions is written at the primitive level.
These primitive blocks comprise the SPROCcells function library. They are
optimized for the hardware and efficiently implemented to extract maximum
performance from the SPROC chip. Other primitive blocks include the glue
blocks or phantoms required to provide control and synchronization functions
for the multiple general signal processors (GSPs) on the SPROC chip.

Detailed Description Text - DETX (338):
The Schedule module takes the top-level SDL file and breaks the file apart
based on the resource and synchronization requirements of the blocks within the
file. Resource requirements include program memory usage, data memory usage,
and GSP cycles. Synchronization requirements include the determination of how
and when blocks communicate data, and whether a block is asynchronous and
independent of other blocks in the design.

Detailed Description Text - DETX (339):
After breaking up the file to accommodate resource and synchronization
requirements, the Schedule module partitions the file by blocks and locates the
blocks to execute on the multiple GSPs on the SPROC chip using a proprietary

partitioning algorithm. The module inserts phantom blocks as necessary to
control the synchronization of the GSPs as they execute the design.

Detailed Description Text - DETX (359):
Time Zones: A time zone declaration is required for every signal source
block, like the primitive block for a signal generator function, or a serial
input port function. The time zone statement declares a time zone name for the
block, and optionally, a sample rate for that zone, in samples per second. A
sample rate need only be given in one such time zone statement of multiple
blocks that declare the same time zone name. The time zone name is used to
determine synchronization requirements of blocks. Time zones which have
different names are taken to be asynchronous, even if they have the same sample
rate.

Detailed Description Text - DETX (418):
Returning to details of the scheduler/compiler 2040, the basic function of
the scheduler/compiler 2040 is to take the user's design which has been
translated into a scheduler/compiler understandable format (e.g., SPROC
Description Language), and to provide therefrom executable SPROC code (.spp),
initial data values (.spd), and the symbol file (.sps). The preferred code for
the scheduler/compiler is attached to the disclosure of U.S. Pat. No.
5,287,511 as Appendix M, and will be instructive to those skilled in the art.

Detailed Description Text - DETX (420):
The scheduler/compiler 2040 also insert "phantom blocks" into the user's
design which supply the necessary system "glue" to synchronize processors and

input/output, and turn the user's design specification into executable code to effect a custom operating system for the design. Preferred code for the phantom blocks is found attached to the disclosure of U.S. Pat. No. 5,287,511 as Appendix E (Scheduler/Compiler Phantom Block Source Code) incorporated herein by reference.

Detailed Description Text - DETX (421):
Because it is possible for a block which the user has designated to have a varying execution time, the GSPs running common code under temporal partitioning could conceivably collide or get out of sequence. Phantom blocks called "turnstiles" are inserted at every sample period's worth of code to keep the GSPs properly staggered in time. By computing and using maximum and minimum durations rather than a maximum duration and an assumed minimum duration of zero, the turnstiles may be placed to optimize the code variability. The scheduler/compiler code provided in Appendix M attached to the disclosure of U.S. Pat. No. 5,287,511, however, does not optimize in this manner. Also, output FIFOs are created whose size depends on code execution time variability. These output FIFOs can also be optimized.

Detailed Description Text - DETX (427):
A high level flow diagram of the compiler preferably used in conjunction with the SPROC 10 of the invention is seen in FIG. 12. When the user of the development system wishes to compile a design, the user runs the compiler with an input file containing the design. The compiler first determines at 1210 which of its various library blocks (cell library 2015) are needed. Because some of the library blocks will need sub-blocks, the

compiler determines at
1212 which sub-blocks (also called child blocks) are required and whether all
the necessary library block files can be read in. If they can, at 1220 the
compiler creates individual instances of each block required, since the same
block may be used more than once in a design. Such a block may be called with
different parameters which would thereby create a different version of that
block. The instances generated at step 1220 are represented within the
compiler data structures as a tree, with the top level block of the user's
design at the root of the tree. At 1230, the compiler evaluates the contents
of each instance, and establishes logical connects between the inputs and
outputs of child instances and storage locations in higher level instances. In
evaluating an instance, the compiler determines code and data storage
requirements of that instance, and assembles the assembly language instructions
which comprise the lowest level child instances. At 1240, the compiler
sequences the instances by reordering the list of child instances contained in
each parent instance. This is the order in which the set of program
instructions associated with each lowest level child instance will be placed in
the program memory 150 of the SPROC 10. To do this, the compiler traces
forward from the inputs of the top level instance at the root of the tree,
descending through child blocks as they are encountered. When all inputs of an
instance have been reached, the instance is set as the next child instance in
the sequence of its parent instance. Feedback loops are detected and noted.
At 1250, the compiler partitions the design over multiple GSPs. Successive
child instances are assigned to a GSP until adding one more instance would
require the GSP to take more than its allowed processing

time; i.e. one sample
period. Succeeding child instances are assigned to a new GSP, and the process
continues until all the instances are assigned to respective GSPs. As part of
the partitioning step 1250, the compiler inserts instances of phantom blocks at
the correct points in a child sequence. Phantom blocks are blocks which are
not designated by the user, but which are necessary for the correct functioning
of the system; e.g. blocks to implement software FIFOs which pass signals form
one GSP to the next GSP in the signal flow. At step 1260, the compiler
re-evaluates the instances so that the phantom block instances added at step
1250 will be fully integrated into the instance tree data structure of the
compiler. Then, at 1270, the compiler generates program code (.spp) by
traversing the instance tree in the sequence determined at step 1240, and when
each lowest level child instance is reached, by outputting to a file the
sequence of SPROC instructions assembled for that instance. It also outputs to
a second file desired initialization values (.spd) for the data storage
required at each instance. It further outputs to a third file the program and
data locations referenced by various symbolic names (.sps) which were either
given by the user or generated automatically by the compiler to refer to
particular aspects of the design. As aforementioned, additional details of the
scheduler/compiler may be seen in the preferred code for the scheduler/compiler
which is attached to the disclosure of U.S. Pat. No. 5,287,511 as Appendix M.

Detailed Description Text - DETX (488):
With the block diagram so provided on an OrCad graphic interface, and in
accord with the above description, after translation by the MakeSDL file, the

scheduler/compiler provides a program file (yhpdual.spp) and a data file
(yhpdual.spd) for the SPROC, and a symbol file (yhpdual.sps) for the symbol
translator and microprocessor and for the SPROCdrive interface. The program,
data, and symbol files are attached to the disclosure of U.S. Pat. No.
5,287,511 as Appendices F, G, and H and incorporated herein by reference. In
addition, the yhpdual.spp and yhpdual.spd files are processed by the MakeLoad
program which generates the yhpdual.blk file which is attached to the
disclosure of U.S. Pat. No. 5,287,511 as Appendix I incorporated herein by
reference.

Detailed Description Text - DETX (489):

In order to completely implement the low frequency impedance analyzer such
that it may be accessed by the microprocessor, the microprocessor is provided
with C code. An example of C code (Maintest.C) for this purpose is attached
hereto as Appendix J. Of course, similar code could be generated in an
automatic fashion if an automatic microprocessor code generator were to be
utilized. As provided, the C code attached to the disclosure of U.S. Pat.
No. 5,287,511 as Appendix J calls yhpdual.c and yhpdual.h which are the
translated files generated by the symbol translator from the yhpdual.spp file
generated by the SPROC scheduler/compiler. Attached to the disclosure of U.S.
Pat. No. 5,287,511 as Appendices K and L are the yhpdual.h and yhpdual.c
files. Thus, the Maintest.C as well as the yhpdual.h and hypdual.c files are
provided in a format which can be compiled by the microprocessor compiler.

Related Application Patent Number - RLPN (1):

5287511

US Reference Patent Number - URPN (5):
5287511

US Reference Group - URGP (5):
5287511 19940200 Robinson et al. 395/700